SEQUOIA: scalable policy-based access control for search operations in data-driven applications

Jasper Bogaerts, Bert Lagaisse, and Wouter Joosen

imec-DistriNet, KU Leuven, 3001 Leuven, Belgium
first.last@cs.kuleuven.be

Abstract. Policy-based access control is a technology that achieves separation of concerns through evaluating an externalized policy at each access attempt. While this approach has been well-established for request-response applications, it is not supported for database queries of data-driven applications, especially for attribute-based policies. In particular, search operations for such applications involve poor scalability with regard to the data set size for this approach, because they are influenced by dynamic runtime conditions. This paper proposes a scalable application-level middleware solution that performs runtime injection of the appropriate rules into the original search query, so that the result set of the search includes only items to which the subject is entitled. Our evaluation shows that our method scales far better than current state of practice approach that supports policy-based access control.

1 Introduction

Access control is a crucial security measure constraining actions that subjects (e.g., users) can perform on resources. To manage this, several requirements must be taken into account. These include the ability to specify fine-grained rules and support for separation of concerns [6], which enables application developers to delegate security management responsibilities.

A combination of policy-based and attribute-based access control satisfies these requirements. Policy-based access control externalizes access control from the application code and has a policy engine evaluation at each access attempt [21]. This technology provides separation of concerns and increases application modularity. Attribute-based access control supports attributes to be assigned to subjects, actions, resources and the environment. These attributes are compared to each other and to concrete values to determine if access is permitted [11]. This supports specification of fine-grained rules such as *"a document can be read by its creator at any time, and by members of the IT department during working hours"*. XACML [14] is considered the de-facto standard policy language for policy-based, attribute-based access control, with characteristics such as policy trees and multi-valued logic.

Because databases hold a crucial position within IT infrastructures, support for properties such as the ability to enforce fine-grained rules and separation

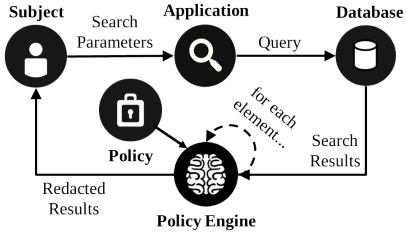

Fig. 1: The a posteriori filter approach evaluates an externalized policy for each item of the result set.

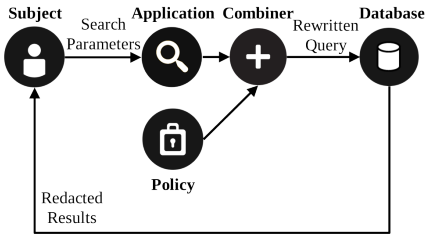

Fig. 2: The rewriting approach takes into the access control policy as part of the query.

of concerns is essential for database operations performed by data-driven applications as well. However, existing approaches generally scale insufficiently with regard to the database size.

Access control techniques integrated in **database systems** fall short for three reasons. First, they require database administrators to be involved in the specification of the policies, thereby violating the separation of concerns. Second, contemporary applications are designed according to a multi-tier architecture. This results in applications that perform queries on behalf of the subject without the latter being identified to the database, which only supports access to be constrained for individual applications instead of for subjects [20]. Third, in large scale deployments such as cloud applications, subjects are managed at the application level and not by the identity management system of the database.

In contrast, access control techniques in the **application**, such as an *a posteriori filter* approach can support externalized policies, but evaluate them for each item that is part of the search result (i.e., the *resources*). This approach filters out any item to which the subject is not entitled based on a policy evaluation, as illustrated in Figure 1. While this approach supports separation of concerns, as security administrators can manage policies independently from the application, it does not scale with an increasing result set. This is true especially for large attribute-based policies [22].

This paper takes an alternative approach that performs runtime injection of the appropriate access rules into the search query based on the context in which subjects perform the search operation. It is illustrated in Figure 2. The approach leverages the filtering system of the underlying database to select only items to which the subject is entitled. Using this approach, we support separation of concerns, the ability to specify attribute-based policies, and can scale with regard to the database size. This paper presents the following contributions:

- A set of well-defined transformation rules that rewrites STAPL policies [2], which are similar to XACML [14], into search queries for RDBMSes.
- An architecture and evaluation of Sequoia, an application-level middleware that transforms and executes search queries for data-driven applications.

This paper is organized as follows: Section 2 describes supporting technologies and discusses the state of the art. Section 3 elaborates on the architecture of Sequoia that enables query rewriting. Section 4 discusses how STAPL policies can be transformed to a query expression. Section 5 provides an evaluation of a prototype of Sequoia. Section 6 concludes the paper.

2 Background and related work

This section discusses the background that serves as a basis for the remainder of the paper. First, it discusses the supporting technologies, such as XACML and STAPL, and provides further analysis of the problem. Next, it elaborates on related database access control technologies.

Supporting technologies. Access control policies can be externalized from the application into a separate artifact that is evaluated by a specialized engine at each access attempt [21]. As opposed to in-code access control, this approach, called *policy-based access control*, increases modularity, avoids application redeployment when a policy is modified, and provides separation of concerns.

XACML [14] is a framework and policy language that supports policy-based access control. It also supports the specification of *attribute-based policies*, which enables fine-grained rule specification. Attribute-based policies support attributes assigned to subjects, resources, actions and the environment that are compared to each other and to concrete values in *expressions*. Attributes are substituted by concrete values at each access attempt to determine if access is permitted.

The basic elements of a XACML policy are *policy components* and *rules*[1]. Rules have a *condition* expression, and policy components have a *target* expression[2]. An expression evaluates to `true`, `false`, or leads to an error (e.g., when an attribute could not be retrieved). Expressions compare attributes or combine other expressions with logical operators (i.e., and, or, not). Whenever an expression evaluates to `false` for a rule or policy component, that element is *not applicable*. Elements that are not applicable are not taken into account in the evaluation decision. For example, rule r_1 of Figure 3 is not applicable for a subject with `org=bankA` that performs a `view` action on a resource with `is_private=true` and `destination_org=bankB`. However, the targets of p_1 and p_2 are applicable for this access attempt.

Besides a condition, rules also specify an effect (i.e., `permit` or `deny`) that is taken into account when the condition of the rule is applicable. As a result, policy elements evaluate to permit, deny, not applicable or an error (called indeterminate in XACML).

A policy component evaluation can yield multiple, possibly conflicting decisions (e.g., when multiple rules are applicable). This is resolved by using *combining algorithms*. This paper focuses on permit overrides (in which a permit

[1] XACML differentiates between policy *sets* and policies, but for brevity we make no distinction in this paper.

[2] Rules can also have targets, but a conjunction with their condition is semantically equivalent, so we disregard this.

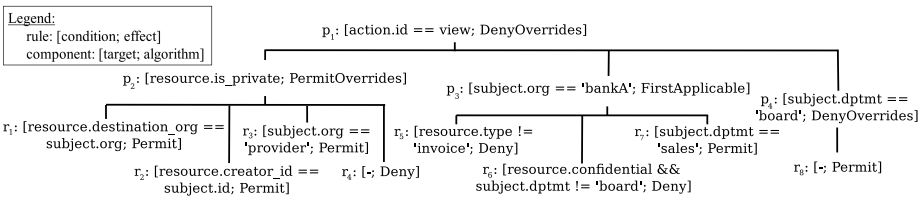

Fig. 3: Example of a XACML-like policy. Policy component p_1 has a deny overrides combining algorithm and a target expression that specifies its children are only relevant for *view* actions. Components p_2 and p_3 have a permit overrides and first applicable combining algorithm, respectively, and child rules. Dashes indicate empty expressions.

decision overrides other decisions of direct children in the component), deny overrides and first applicable (in which the first applicable rule determines the final decision). For example, p_1 in Figure 3 has a deny overrides algorithm, meaning that if p_3 evaluates to deny, then p_1 evaluates to deny regardless of the decision of p_2 or p_4. Policy components have either rules or other policy components as children, thus forming *policy trees*. When their target expressions are applicable, their child elements are also evaluated to come to a policy decision.

This paper uses STAPL [2] policies as a basis for transformation. STAPL is a framework and policy language that closely resembles XACML, but which offers more ease of use and a slightly better evaluation performance. Any XACML policy can generally be converted to a STAPL policy, and STAPL policies can similarly be expressed in XACML. Hence, the transformation process applied in this paper is also applicable for XACML policies.

Figure 3 illustrates a small example policy of an industry case study that motivated this work [7]. Here, if the action that is performed is *"view"* (e.g., as is the case in a search operation), subjects can access documents depending on their department, whether they have created it or were addressed, and so on. For example, a subject of the sales department can view invoices if the organization to which he/she belongs was addressed (rules 1, 5 and 7), unless it involves a confidential invoice (rule 6).

While both XACML and STAPL support fine-grained specification of access rules, their policy evaluation process also involves a considerable overhead [22]. For traditional request-response applications, this overhead is in many cases acceptable. When the policy is evaluated for a large set of resources such as for search operations on a database, however, this can become an impeding factor.

Related work. Because of its importance, a lot of prior research has focused on database security [3]. This paper focuses on row-level access control [3] that follows the Truman model[3] [19]. In this regard, there generally exist three approaches to provide scalable search queries on databases.

The first approach involves techniques generally classified under the term *"Fine-Grained Access Control (FGAC)"* [19]. FGAC uses query rewriting tech-

[3] In a Truman model, queries are transparently modified to restrict access of a subject to database items.

niques [1, 5, 8, 10, 15, 19] to provide database security. This is typically performed using rules that are specified in the native query language of the target database and may be realized through the creation of *views*. While this approach scales with regard to the size of the result set, it also has some issues. In particular, since rules are specified in the native query language, they at least partly violate the principle of separation of concerns [6], because the database administrator must help specify the policies. Worse, this approach generally assumes a two-tier architecture in which subjects directly query the database and can be identified accordingly. Contemporary applications are typically designed according to a multi-tier architecture, and generally perform queries on behalf of the subject without identifying them to the database [20]. This only supports access control to filter based on accessing applications. Moreover, in large-scale deployments such as cloud applications, subjects are typically managed at the application level, and not by the identity management system of the database. Our approach does not suffer these issues due to substitution of subject properties in the query and the support for externalized policies that are rewritten.

A second approach involves configuring the access control component that is used by the database based on an external policy. Compared to the previous approach, this does support separation of concerns. Notable examples for this approach are MyABDAC [12] and a recent system proposed by Mutti et al [13]. MyABDAC uses XACML policies to generate access control lists for the underlying database system. These can then be employed to constrain access. While this approach maintains separation of concerns, it also assumes a two-tier architecture and hence suffers the same issues as the first approach. Moreover, it does not scale well with regard to the size of the database. Mutti et al. introduce a system that extends SQLite for SELinux support. While this system scales, it requires specification of database hooks and supports only lattice-based policies, which are not as fine-grained as XACML policies supported by our approach.

A third approach involves evaluating the access control policy for each of the items in the result set of a search query. Bouncer [16] takes this approach for the CPOL trust management system. While this approach supports separation of concerns and the specification of fine-grained policies, it can also introduce a considerable overhead when the size of the search result set increases. This is especially true for fine-grained rules, typically included in attribute-based policies [22]. Our approach does not suffer from this problem.

This paper pursues an approach that uses query rewriting by using an externalized policy that is injected at runtime as part of the search query. This approach scales with regard of the result set and regards the separation of concerns principle. Also, it does not require DBMS modification. This approach has also been explored by Axiomatics Data Access Filter [18]. Compared to them, this work focuses on the transformation process, including conversion of XACML concepts such as policy trees, combining algorithms and multi-valued logic to a database query. Moreover, we present a thorough evaluation of the approach.

Besides access control techniques, several other security measures can secure search queries on database systems. In particular, secure query processing [24]

and homomorphic encryption [9] aim at supporting queries on an encrypted data set. While these measures have several issues such as performance and fine-grainedness of the data set, they can be used complementary to our approach.

Lastly, this research was also influenced by related work done in the enforcement of usage control. In particular, Pretschner et al. [17] have presented an architecture that enforces usage control policies in a distributed system, with an emphasis of reducing policy enforcement overhead. In contrast, our approach does not focus on distributed evaluation for enforcing policies.

3 Sequoia architecture

This section describes an application-level solution that supports access control on database search operations in data-driven applications. The solution provides scalability, expressiveness and separation of concerns. In order to comply with these requirements, we employ a policy-based, attribute-based access control system as a basis for the access control rules that must be supported. Since XACML is considered the de-facto standard for such access control systems, we use its language model as a basis for the transformation. We use a rewriting approach that involves transforming the policy to a query. The transformation process must cope with fundamental issues to support STAPL policy conversion. In particular, characteristics such as policy trees, multi-valued logic and combining algorithms must be translated to a semantically equivalent query expression.

Scope. This paper analyzes the query rewriting approach for relational databases. We expect resources to be represented as the *rows* of (one or more) tables and their corresponding attributes as the *columns* of these tables. These resources are referred to as *items*. We assume that all attributes of the resources referred to in the policy are stored in the database. In addition, the database schema is expected to provide proper mapping of the attributes onto columns.

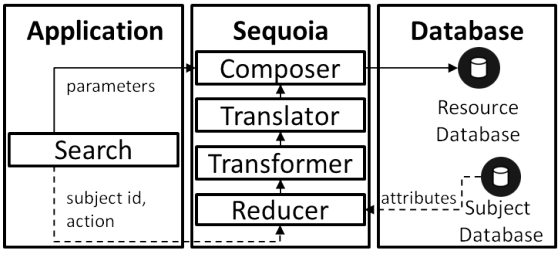

Fig. 4: The rewriting approach reduces, transforms and translates the policy and combines it with the original search parameters into a query.

Overview. Figure 4 provides an overview of the architecture. The Sequoia middleware operates between the application layer and the database. Whenever the application queries the database, the middleware intercepts the query and

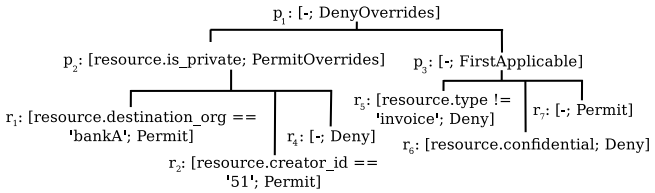

Fig. 5: Reduction of Figure 3 for a subject with `id=51`, `dptmt=sales` and `org=bankA`.

performs run-time injection of the appropriate access rules that are combined with the original search parameters in a query. The composed query reflects only relevant rules for the subject and ensures that only the items to which the subject is entitled (i.e., those permitted if the policy is evaluated for each item of the original search result) are returned.

The policy semantics are preserved throughout the transformation process. The result set for executing the transformed query is equivalent to the set of items permitted when performing a policy evaluation on each item resulting from the original query. To support this, the middleware uses four components: the reducer, transformer, translator and composer.

Reducer. This component obtains the relevant attribute values associated with the acting subject, action and environment. Next, similar to [18], it performs a partial substitution and evaluation of all expressions in the policy that do not refer to any resource attributes (which will be queried in the database). This enables pruning of the policy for rules and policy components that always evaluate to true or false. The reduction minimizes the query that is generated and eliminates the need for subject attributes to be stored in the same database as the resources that are searched. For example, consider the policy in Figure 3 reduced for the `view` action and a subject with `id=51`, `dptmt=sales` and `org=bankA`. This is shown in Figure 5. The policy is significantly smaller, which avoids redundant checks and simplifies the final query.

Transformer. This component transforms the policy to a boolean expression that can be translated to the query language and combined with search parameters at a later stage. Due to policy reduction, no attributes associated with the subject, action or environment should be left in the policy as they were substituted with the relevant values.

The transformation must be equivalent for `permit` decisions. This means that whenever a policy evaluation leads to a `permit` decision for a certain item, the corresponding boolean expression evaluation must also be `true`. In contrast, if the evaluation of a policy leads to a `not applicable` or `deny` decision, its corresponding expression must evaluate to `false`. This also filters out any items for which the evaluation is indecisive. If an error occurs during the evaluation of the boolean expression, the search query must be aborted altogether[4].

[4] In this transformation, we do not consider extended indeterminate decisions that are defined in XACML 3.0. Contrary to request-response applications, errors can not always be gracefully handled for individual data rows.

For example, consider again Figure 5 the reduced policy. The transformation of this policy for the acting subject results in boolean expression $[resource.is\_private \wedge (resource.destination\_org == `bankA' \vee resource.creator\_id == `51')] \wedge [resource.type == `invoice' \wedge \neg resource.confidential]$. This expression is only satisfied for items to which the subject is entitled.

Transformation of STAPL policies is not trivial. In particular, we need to take into account policy trees, multi-valued logic and combining algorithms when a policy is transformed to a single expression. For this reason, we elaborate on this in more detail in Section 4.

Translator. This component translates the expression that resulted from the transformation to the query language of the database to which it is submitted. This involves two tasks. First, the syntax of the expression is translated to the syntax of the query language for the target database. Second, all attributes referenced in the expression are translated to column references in the database.

While the first task is generally straightforward because SQL supports boolean expressions equivalent to the ones to which we transformed, an attribute-to-schema mapping is required for the second task. This mapping describes how attributes are mapped onto the columns corresponding to tables in the database. In some cases, some of the attributes associated with the resource could be stored in a different table than the one that is queried. As a consequence, the mapping enables the translator to cope with the complexity of the database schema and can indicate which joins are required in the search query. The table that is being queried is the *base table* from which joins to auxiliary tables are accommodated.

Composer. This component combines search parameters of the original query with the translated access rules. For example, the transformation of the policy of Figure 5 was previously translated to database-specific syntax. This is now combined with the query using an *and*-operator. Also, any required join operations are injected in the search query.

4 Transformation

This section addresses the second step in the approach outlined in the previous section. It performs the transformation of characteristics such as policy trees, multi-valued logic and combining algorithms to a boolean expression. To achieve a boolean expression, we iterate over two steps until the policy consists of a single policy component with a permit overrides algorithm, no target and only rules as children. We call this a *flat component*. Figure 6 shows an overview of the process. Figure 7 illustrates an example.

As a **first step**, we transform every component in the policy tree that has only rules as children (i.e., a *leaf component*) to an equivalent flat component. Also, we conjunct the target expression of each transformed policy component with the condition expressions of all of its child rules. For this, the original combining algorithm determines how the transformation is performed to retain semantic equivalence. As a **second step**, we regard every *node component*, i.e. components with only flat components as children. We pull up all rules of the

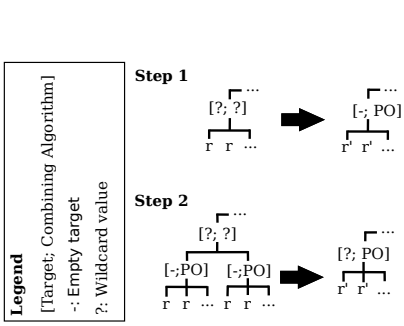

Fig. 6: Transformation process overview

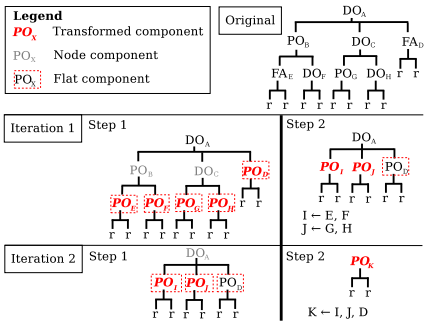

Fig. 7: Policy transformation example

children of these node components and transform them to a policy component with a permit overrides algorithm. Similar to the first step, semantic equivalence is retained through transformation methods that differ for each combining algorithm. By iterating over these steps, the policy tree is gradually flattened to result in a single, flat component containing only rules. A boolean expression is then created as disjunction of the conditions of all permit rule children. Figure 7 illustrates the transformation process applied on a policy. As the figure shows, the policy is gradually transformed to a single component in bottom-up fashion.

The boolean expression is semantically equivalent to the original with regard to the permit decision. In other words, if the policy evaluates to `permit`, the boolean expression evaluates to `true`. For `not applicable` and `deny` decisions in the policy, the boolean expression evaluates to `false`. If an error occurs (e.g., an attribute value could not be retrieved), the search query will be aborted. We have developed a formal proof that the transformation process maintains the semantics of the policy when these equivalence rules are taken into account [4]. Note that Turkmen et al. [23] also developed a flattening process for policies, but focus on policy analysis, while our approach is optimized for database queries.

In the remainder of this section, we elaborate on the two steps that are iterated over during the transformation process. Section 5 evaluates a Sequoia prototype that employs this process for generating queries from STAPL policies.

Step 1: Transforming leaf components

This step transforms all *leaf components* (i.e., policy components with only rules as children) into flat components. In order to retain semantic equivalence, this requires a different transformation approach depending on the combining algorithm of the component. For example, consider component p_3 from Figure 5. When this component is transformed to a flat component, the condition of rule r_7 is transformed so that it evaluates to false if either r_5 or r_6 is applicable.

As a first part of this step, for each rule, the target expression is included in conjunction with the condition of that rule. This is done regardless of the

combining algorithm. Next, we perform a transformation that depends on the combining algorithm of the component[5].

For **first applicable** components, a permit rule is applicable only if none of the deny rules that *precede* it are applicable. Consequently, we can transform a permit rule by conjunction of its original condition with the negation of conditions of its preceding deny rules. The deny rules of the original component can then be included without modification. More formally, the condition of each permit rule r_i of the original leaf component P is transformed as follows:

$$cond(r_i) \wedge \bigwedge_{r_j \in pre(r_i, P)} \neg cond(r_j)$$

In which `cond` indicates the condition of a rule, and `pre` the set of all preceding deny rules in the same component P.

For **deny overrides** components any applicable deny rule overrides other decisions. Consequently, the transformation of a permit rule conjuncts the original condition with the negation of *all* of the deny rule conditions of the original component. Similar to the first applicable transformation approach, deny rules are included without modification. More formally, the condition of each permit rule r_i of the original leaf component P is transformed as follows:

$$cond(r_i) \wedge \bigwedge_{r_j \in deny(P)} \neg cond(r_j)$$

With `deny` the set of all deny rules in the same component P as the given rule.

For **permit overrides** components, no further transformation is required as it already is a flat component after conjunction of conditions with the target.

Consider components p_2 and p_3 from Figure 5 as an example of the approach. In this example, rules r_1, r_2 and r_4 are rewritten to conjunct target *"resource.is_private"* as an additional constraint for their conditions. For p_3, on the other hand, r_7 is transformed to contain a negation of conditions of r_5 and r_6. Also, r_5 and r_6 are included in the result without modification. This results in a permit overrides component with permit rule *"resource.type == 'invoice'* $\wedge$ $\neg$ *resource.confidential"* and original deny rules r_5 and r_6.

Step 2: Pulling up flat components rules

This step involves all policy components that only have flat components as children. The policy components for which all child components satisfy these requirements are called *node components*. In this step, we transform the rules of all child components, and include them as direct children of the node component. We also change the node component combining algorithm to permit overrides.

The transformation approach for pulling up flat component rules to be combined at a higher level in the policy tree differs from the one introduced in step 1. In particular, it takes into account how rules of one child component affect the

[5] For brevity, we do not elaborate on approaches for minimizing the generated expression in this paper.

decision process at the level of the node component. For example, consider p_1 in Figure 5. A deny rule such as r_5 can affect the decision process even if r_1 is applicable, because the deny decision that may stem from r_5 overrides any permit decision due to the deny overrides of p_1.

In order to pull up the rules associated with flat components, the transformation method needs to ensure that the result has the same semantics as the original policy component. Hence, the transformed rules must take into account their original condition, but may also include conditions of rules from other child components of the node component.

If the node component has a **first applicable** algorithm, all of the deny rules of preceding child components are taken into account when a permit rule of a certain child component is transformed. Consequently, the generated rule cannot be applicable if a deny rule of a preceding child component was applicable. This is done by generating a permit rule that has the condition of the original permit rule in conjunction with the negation for each deny rule of preceding components. Similar to the previous step, the deny rules are included without modification. More formally, the condition of each permit rule r_i of a child component is transformed as follows:

$$cond(r_i) \wedge \bigwedge_{r_j \in preco(r_i)} \neg cond(r_j)$$

In which `preco` reflects all deny rules from policy components of the given rule's ancestor that precede its parent. Deny rules are included in the result without modification. Note that if a deny rule of `preco` is applicable, it can still be overridden by a permit rule condition of its own component.

If the node component has a **deny overrides** algorithm, the permit rules of all child components are combined in a single, *unified* permit rule. All deny rules are again included without modification. The unified rule must ensure two properties. First, at least one permit rule of the child components must be applicable in order for the unified rule to be applicable. Second, for each child component, if no permit rule is applicable in that child, then its deny rules must not be applicable. Otherwise, the unified rule is also not applicable. The first property ensures that the *not applicable* decision is propagated during the flattening of the policy tree. The second property ensures that if a child component leads to a deny decision, then the evaluation of its parent component will also lead to a deny decision because of the deny overrides algorithm of the node component. In this case, the unified rule can not be applicable, while a deny rule is. The unified permit rule has a condition that is a conjunction of a clause of all permit rule conditions `appl`, and all *component clauses* `comp`. Here, `appl` ensures the first property and the component clauses ensure the second property of the unified rule. More formally, we describe this as

$$appl(P) := \bigvee_{r_i \in permits(P)} cond(r_i)$$

In which `permits` retrieves all permit rules of the given policy component and all its children. Similarly, `denies` fetches deny rules of a component and its children. For each child component P_C, component clause `comp` is constructed as

$$comp(P_C) := (\bigvee_{r_i \in permits(P_C)} cond(r_i)) \vee \bigwedge_{r_i \in denies(P_C)} \neg cond(r_i)$$

If a node component has a **permit overrides** algorithm, all rules of its children are included without modification.

For example, consider again component p_1 from Figure 5. In order to be permitted, the following expression must be satisfied for a evaluation request: *"*$[resource.is\_private \wedge (resource.destination\_org == `bankA' \vee resource.creator\_id == `51')] \wedge [resource.type == invoice \wedge \neg resource.confidential]$*"*. Here, the unified rule was simplified by applying common logical reduction techniques.

Flat component to boolean expression

The transformation steps are repeated until a single, flat component is resulted. Finally, the flat policy component is converted to a boolean expression by constructing a disjunction of all permit rule conditions of the flat policy component. If none of the permit rule conditions are applicable, the boolean expression will evaluate to false, and the database item for which the expression was evaluated will not be included in the result set. This maintains the original policy semantics because each transformation step ensures that a permit rule of the transformation is only applicable if it was not overridden by a deny decision. A formal equivalence proof is given in [4].

5 Performance evaluation

As discussed earlier in this paper, the a posteriori filter approach for supporting policy-based access control for search operations on databases becomes prohibitive with regard to performance when the size of the search result set is large. To resolve this, we have presented an alternative approach that uses query rewriting to reduce the overhead of access control. This section evaluates an implementation of the rewriting approach discussed in Section 3 and Section 4 for transformations from STAPL [2] policies to relational databases, and compares it to the a posteriori filter approach that was illustrated in Figure 1.

We evaluate five aspects. First, we discuss scalability of the rewriting approach compared to the a posteriori filter. Second, we evaluate the overhead introduced by the rewriting approach. Third, we discuss the impact of policy size for different combining algorithms. Fourth, we inspect the performance impact of the amount of evaluated attributes. Fifth, we also determine the impact of the proportion of permitted items on performance.

Setup. The evaluation was performed on a Dell OptiPlex 755 computer with Intel Core 2 Duo 3GHz processor and 4GB internal memory using Ubuntu 15.10 as an operating system and performing all database requests using JDBC to a MariaDB [6] database with caching disabled as much as possible. All queries were performed locally (i.e., no network traffic was involved). We repeated all tests 10000 times after 100 warmups and took the mean values for processing times.

[6] `https://mariadb.org/`

Scalability. As a first part of the evaluation, we have assessed the scalability of the rewriting approach with regard to the size of the search result set. To do this, we have compared how the approach performs with regard to a posteriori filter, which was illustrated in Figure 1. The test evaluated the total processing time required to perform the search query together with determining what items must be part of the result set. We did this based on an extensive policy that was inspired by an industry case study [7] that motivated this work and that contains 32 policy components and 63 rules that regard 33 attributes[7].

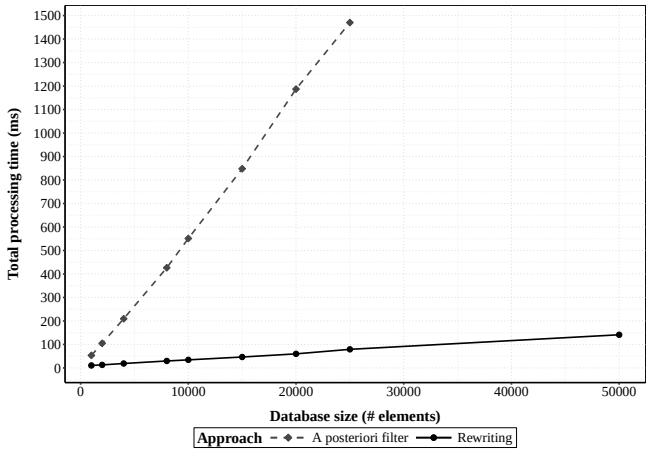

Fig. 8: Comparison of processing times for an increasing data set size. Lower is better.

The results are shown in Figure 8. They demonstrate the processing times involved with both approaches for a subject that is entitled to about 40% of the items. As the figure shows, the rewriting approach performs far better than the a posteriori filter. We only performed evaluation for the latter for up to 25000 items, because measuring it involved too much processing time for repeated tests. In contrast, the rewriting approach requires considerably less processing time with regard to the database size. For example, the same test for a data set of a million items had a processing time of 4037ms. This included the serialization of the items in the result set in Java data structures, which amounted to 67% of the query processing time on average. Consequently, we can conclude that the rewriting approach scales well with an increasing data set.

Overhead. As a second part of the evaluation, we inspected the overhead involved with the different steps of the rewriting process. We have done this for an increasing search result size in a same test setup as the scalability test.

Figure 9 shows the overhead of different transformation steps in the rewriting approach. The figure shows that the policy reduction and transformation steps

[7] This policy can be found at `https://github.com/stapl-dsl/stapl-examples/blob/master/src/main/scala/stapl/examples/policies/EdocsPolicy.scala`.

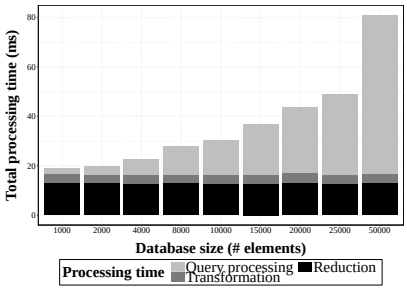

Fig. 9: Transformation overhead analysis. The processing times for reduction and transformation steps remain constant, whereas query processing times increase with the amount of data items.

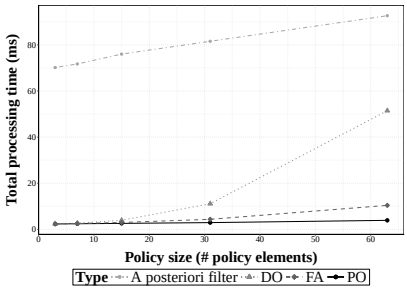

Fig. 10: The processing time in function of the policy depth for first applicable (FA), deny overrides (DO), permit overrides (PO) policies and the a posteriori filter. Lower is better.

remain constant and are dominated by the query processing for larger result sets. The overhead for the policy reduction and transformation steps depends on the subject. For some subjects, for example, the transformation step amounts to less than 1ms.

Because overhead of the transformation steps remain constant, the query processing time becomes the dominant factor in the total processing time as the size of the database increases. In the a posteriori filter approach, the time required for policy evaluation on each element is the dominating factor for the processing time. Also, note that the result of the transformation can be cached for each subject, which reduces the overall processing time for subsequent searches.

Policy size impact. As a third part of the evaluation, we have assessed the impact of the size of the policy on the processing time of the rewriting approach. We evaluate this for the deny overrides, permit overrides and first applicable combining algorithms.

The test generated policies for varying policy *depths*, i.e., numbers of nodes on the path from the root policy component to the leaf-level components. For each depth, the policy contains $2^{d+1} - 1$ policy elements, 2^d of which are rules. Each component has two children and the same combining algorithm. Leaf components all have one permit and one deny rule. The test was performed on 1000 elements.

Figure 10 shows the processing time for the rewriting approach for different combining algorithms and the a posteriori filter when the depth of the policy tree increases. For the a posteriori filter, we have plotted the mean evaluation time for the three policy types, as they had similar processing times. As expected, total processing times increase due to an increasing number of policy elements. The deny overrides algorithm performs worst for the rewriting approach. This is because the transformation introduces inevitable redundancy in the query to maintain original semantics. This test, however, involves the worst-case scenario for a large policy. Moreover, a posteriori filtering still exceeds the query rewriting approach for an extensive policy. The largest proportion of the overhead is due

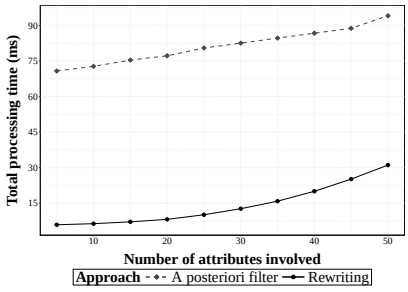

Fig. 11: Processing time in function of the number of involved attributes. Lower is better.

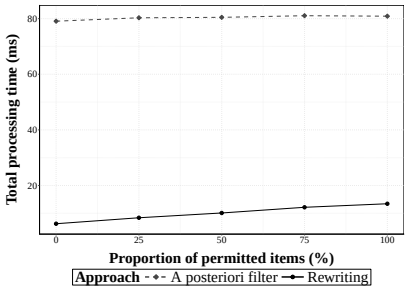

Fig. 12: Impact of the proportion of permitted items on processing times. Lower is better.

to policy transformation, which amounts to about 85% of the total time for deny overrides components in the rewriting approach.

The permit overrides transformations, on the other hand, perform best because the transformation does not need to take into account the conditions of any deny rules.

Attribute impact. As a fourth part of the evaluation, we investigated the impact that the amount of attributes involved in the evaluation had on the processing time of the secured search query. To determine this, we generated several policies that require n resource attributes to be evaluated prior to resorting to a decision, with n ranging from 5 up to 50 attributes. The test was performed on a data set of 1000 items, from which 50% are part of the final result.

Figure 11 shows that the rewriting approach performs better than the a posteriori filter, while difference in processing times is fairly constant. The a posteriori filter has higher overhead because all items are fetched and evaluated. For the rewriting approach, the curve is explained due to the time required to perform the transformation (up to 78% of the total time) and the overall impact of the amount of attributes seems limited for the amount of attributes involved.

Result size impact. As the last part of the evaluation, we measured the impact on the processing time of the proportion of items that are part of the permitted result set. This test considers the processing time for an increasing percentage of items of the data set to which the subject is entitled. For this test, we used the same setup as for the attribute impact evaluation.

Figure 12 demonstrates that the processing time for the a posteriori filter approach remains fairly constant, because it needs to evaluate all items regardless of the proportion of permitted items. In contrast, the rewriting approach has a processing time that is directly proportional to the amount of items that are permitted. In other words, if a subject is entitled to a smaller proportion of the data set, the rewriting approach will perform better than if he/she is entitled to a large proportion. For example, if a subject is entitled to 25% of the items, the processing time will be smaller than if he/she is entitled to 75% of the items. In

all cases however, the rewriting approach performs significantly better than the a posteriori filter.

Summary. The evaluation indicates that the rewriting approach performs far better than the a posteriori filter, because it scales well with regard to the result set size and the number of attributes evaluated. Moreover, the approach performs even better when a subject is entitled to a smaller proportion of the data set, which is common in contemporary applications.

6 Conclusion

This paper presented an application-level middleware that supports scalable policy-based access control for search queries on relational databases. It does this in a manner that also supports expressive policies and separation of concerns. Because the de-facto standard for policy-based, attribute-based languages is XACML, this paper also elaborated on a method that transforms policies with a similar model to a boolean query expression that can be translated and combined with the search parameters. The evaluation shows that our approach performs far better than the current state of practice. As a result, we can conclude that this work constitutes an important step for maturation of both policy-based access control and database access control using application-level middleware.

References

1. Oracle Virtual Private Database (VPD). `http://docs.oracle.com/cd/B28359_01/network.111/b28531/vpd.htm`. Accessed: 2016-09-09.
2. Simple Tree-structured Attribute-based Policy Language (STAPL). `https://github.com/stapl-dsl`, June 2016. Accessed: 2016-09-26.
3. E. Bertino and R. Sandhu. Database security-concepts, approaches, and challenges. *Dependable and Secure Computing, IEEE Transactions on*, 2(1):2–19, 2005.
4. J. Bogaerts, B. Lagaisse, and W. Joosen. Transforming xacml policies into database search queries. Technical report, KU Leuven, 2017.
5. B. Carminati, E. Ferrari, J. Cao, and K. L. Tan. A framework to enforce access control over data streams. *ACM TISSEC*, 2010.
6. B. De Win, F. Piessens, W. Joosen, and T. Verhanneman. On the importance of the separation-of-concerns principle in secure software engineering. In *Application of Engineering Principles to System Security Design*, 2002.
7. M. Decat, J. Bogaerts, B. Lagaisse, and W. Joosen. The e-document case study: functional analysis and access control requirements. Technical report, KU Leuven, 2014.
8. S. Franzoni, P. Mazzoleni, S. Valtolina, and E. Bertino. Towards a fine-grained access control model and mechanisms for semantic databases. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 993–1000. IEEE, 2007.
9. C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.
10. E. Grummt and M. Müller. Fine-Grained Access Control for EPC Information Services. In *The Internet of Things*. Springer, 2008.
11. V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. Guide to Attribute Based Access Control (ABAC) Definition and Considerations. *NIST Special Publication*, 2014.

12. S. Jahid, C. A. Gunter, I. Hoque, and H. Okhravi. MyABDAC: compiling XACML policies for attribute-based database access control. In *Proceedings of the first ACM conference on Data and application security and privacy*, pages 97–108. ACM, 2011.
13. S. Mutti, E. Bacis, and S. Paraboschi. SeSQLite: Security Enhanced SQLite: Mandatory Access Control for Android databases. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 411–420. ACM, 2015.
14. OASIS. eXtensible Access Control Markup Language (XACML) Standard, Version 3.0. `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf`, 2013.
15. L. E. Olson, C. A. Gunter, W. R. Cook, and M. Winslett. Implementing reflective access control in SQL. In *Data and Applications Security*. Springer, 2009.
16. L. Opyrchal, J. Cooper, R. Poyar, B. Lenahan, and D. Zeinner. Bouncer: Policy-Based Fine Grained Access Control in Large Databases. *International Journal of Security and Its Applications*, 2011.
17. A. Pretschner, M. Hilty, and D. Basin. Distributed usage control. *Communications of the ACM*, 2006.
18. E. Rissanen. Fine-grained relational database access-control policy enforcement using reverse queries, May 19 2015. US Patent 9,037,610.
19. S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD*. ACM, 2004.
20. A. Roichman and E. Gudes. Fine-grained access control to web databases. In *Symposium on Access control models and technologies*. ACM, 2007.
21. P. Samarati and S. Vimercati. Access Control: Policies, Models, and Mechanisms. In *Foundations of Security Analysis and Design*, pages 137–196. 2001.
22. F. Turkmen and B. Crispo. Performance evaluation of XACML PDP implementations. In *Proceedings of the 2008 ACM workshop on Secure web services*, 2008.
23. F. Turkmen, J. Hartog, S. Ranise, and Z. N. Analysis of XACML Policies with SMT. In *4th Conference on Principles of Security and Trust (POST)*, 2015.
24. S. Wang, D. Agrawal, and A. El Abbadi. A comprehensive framework for secure query processing on relational data in the cloud. In *Secure Data Management*, pages 52–69. Springer, 2011.